**Mule**

**Flow is a message processing block that has its own processing strategy and exception handling strategy. Used in integration tasks, data processing, connecting applications, event processing, etc.**

**Subflow always processes messages synchronously but inherits processing strategy and exception handling strategy from the calling flow. It can be used to split common logic and be reused by other flows.**

**Private flow does not have source define. It can be synchronous or asynchronous based on the processing strategy selected. Also they have their own exception handling strategy. Allows you to define different threading profile.**

**ForEach**

1. splits a collection into individual elements and processes them iteratively through the processors embedded in the scope, then **returns the original message to the flow**.
2. it can iteratively process elements from any type of collection, including **maps, lists, arrays, and MuleMessageCollection.**
3. You can insert any message processor, except inbound connector in the Foreach scope.If we drag a **two-way connector into a Foreach scope, Mule automatically converts it to an outbound-only connecto**r.
4. When foreach scope receives a message payload that is not a collection type like maps, lists, arrays, and MuleMessageCollection it **throws an IllegalArgumentException.**
5. **The rootMessage** variable(flow Variable) is default associated with the message, contains a reference of the complete, unsplit message collection .
6. **The counter variable** (flow Variable)is default associated with the message , Foreach uses counter variable to record the number of the elements it has processed.

**Filters**

Mule [**Filters**](https://docs.mulesoft.com/mule-user-guide/v/3.7/filters) are a handy way of analyzing the messages passing through our flow and determining whether they should be processed or not. Simply put, once a message reaches a filter component in the flow and meets the filter’s logical conditions, the message may pass through untouched. Otherwise, flow execution is terminated and no more processing is done. Mule offers several filters which range from implementing logic operators (**and**, **or** and **not**) to validating input against a ReGex or schema, and also allows us to create a [**custom filter**](https://docs.mulesoft.com/mule-user-guide/v/3.7/custom-filter).

One significant consequence of using a filter is the fact that there **is no visible indication that a message has failed to pass through it**. Execution just stops.

In situations where we need to react to these invalid messages, **there are two options available to us**:

1. The first is to use a [**Choice**](https://docs.mulesoft.com/mule-user-guide/v/3.7/choice-flow-control-reference) router component instead.
2. Otherwise, if a filter must be used, the second option is to wrap it in a [**Message Filter**](https://docs.mulesoft.com/mule-user-guide/v/3.7/message-filter).

**<message-filter onUnaccepted="unaccepted-messages-handler-flow" >**

**<expression-filter expression="#['thisString'.equals('thisOtherString')]"/>**

**</message-filter>**

The Message Filter has two important parameters which are used if a messages fails to meet our underlying filter’s condition:

* **onUnaccepted()** – this should be populated with the name of a message processor or flow. Instead of simply stopping processing, the invalid message is redirected to this processor or flow;
* **throwOnUnaccepted** – if set to true, the Message Filter throws an exception on unaccepted messages.

**Types of Filters**

* 1. **Logic Filters(And,Or,Not)**
  2. **Message Property Filter()**

This filter allows you add logic to your routers based on the value of one or more properties of a message. This filter can be very powerful because the message properties are exposed, allowing you to reference any transport-specific or user-defined property. For example, you can match one or more HTTP headers for an HTTP event, match properties in JMS and email messages, and more.

* <message-property-filter pattern="Content-Type=text/xml" caseSensitive="false"/>

### 3) Exception Type Filter

A filter that matches an exception type.

|  |  |
| --- | --- |
| 1 | <exception-type-filter expectedType="java.lang.RuntimeException"/> |

Payload Type Filter

Checks the class type of the payload object inside a message.

|  |  |
| --- | --- |
| 1 | <payload-type-filter expectedType="java.lang.String"/> |

## Custom Filters

**Implements Fillter interface**

**boolean** accept(MuleMessage message);

**Routers (Flow Controls**

**route messages to various destinations in a Mule flow**. Some routers incorporate logic to analyze and possibly transform messages before routing takes place

* Split a message into several segments, then route each segment to a different building block.
* Combine several messages into a single message before sending it to the next building block in the flow.
* Reorder a list of messages before sending it to the next building block.
* Evaluate a message to determine which of several possible building blocks it should be routed to next.
* Broadcast the same message to multiple building blocks.

### Flow Controls That Don’t Modify the Payload

|  |  |
| --- | --- |
| Choice | Evaluates a message against specified criteria, then sends it to the first message processor that matches those criteria. |
| First Successful | Sends a message to the next message processor within a "circular" list of processor targets. |
| Round Robin | Iterates through a list of two or more message processors, sending successive messages to the next message processor on the list. When it reaches the end of the list, it jumps to the start of the list and resumes the iteration. |

### Flow Controls That Modify the Payload

|  |  |
| --- | --- |
| Collection Aggregator | Checks the group tag (known as a **Correlation ID**) attached to each message in a group to create a collection of messages which share the same Correlation ID. |
| Collection Splitter | Accepts a collection of messages (or parts of messages), splits them into individual messages, then sends each new message, in sequence, to the next message processor in a flow. |
| Message Chunk Aggregator | Checks the group tag (Correlation ID) of each message in a collection, selects all the messages whose group tag matches the specified value, then combines those messages into a single message which is then sent to the next message processor in an application flow. This is particularly useful for re-assembling the segments of a long message that has been received as multiple messages, each one consisting of a segment of fixed length created and sent by the Message Chunk Splitter. |

**Scopes**

**Async:** executes simultaneously with the parent message flow. This type of processing block can prove useful for executing time-consuming operations (such as printing a file or connecting to a mail server) - as long as those operations do not require sending a response back to the parent flow.

Since they operate on a copy of the message on a different thread, async scopes cannot, by definition, support request-response exchange patterns. Instead, they must implement one of several supported one-way processing strategies, as detailed in the configuration section, below.

If no processing strategy is configured for the async scope, Mule applies the default processing strategy: **queued asynchronous**

**Shared Resource in Mule:**We can make connectors as an reusable component by defining them as common resources and expose them to all applications deployed under a same domain, these resources are known as shared resources.

**Domain projects – shared resources needs to define inside the Mule Domain Projects and then referred to each of the projects that are meant to use the elements in it.**

**Error Handling:**

Mule provides numerous options for handling errors. Faults that occur within Mule are referred to as **exceptions**; when an activity in your Mule instance fails, Mule throws an exception. To manage these exceptions, Mule allows you to configure exception strategies.

Mule fall into one of two categories:

**1)System Exceptions**, 2) **Messaging Exceptions**.

## System Exceptions

Mule invokes a **System Exception Strategy** when an exception is thrown at the system-level (that is, when no message is involved, exceptions are handled by system exception strategies). For example, system exception strategies handle exceptions that occur:

* During application start-up
* When a connection to an external system fails

When a system exception strategy occurs, Mule sends an exception notification to registered listeners, logs the exception, and — if the exception was caused by a connection failure — executes the [reconnection strategy](https://docs.mulesoft.com/mule-runtime/3.8/configuring-reconnection-strategies). System Exception Strategies are not configurable in Mule.

Java.io.IOException ->java.net.SocketException->java.net. ConnectException

**Example Scenario**

Mule establishes a connection to a JMS broker in order to receive a message. When Mule attempts to use the connection to consume a message the connection fails, which causes Mule to invoke the system exception strategy. Because the failure occurred before any message was received for processing, Mule invoked the *system*, rather than *messaging*, exception strategy.

## Messaging Exceptions

Mule invokes a **Messaging Exception Strategy** whenever an exception is thrown within a flow (i.e., whenever a message is involved, exceptions are handled by messaging exception strategies).

[java.lang.Throwable](http://java.sun.com/j2se/1.6.0/docs/api/java/lang/Throwable.html?is-external=true) -> [java.lang.Exception](http://java.sun.com/j2se/1.6.0/docs/api/java/lang/Exception.html?is-external=true) ->

[org.mule.api.MuleException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/MuleException.html) ->

**org.mule.api.MessagingException**

**Direct Known Subclasses:**

[ComponentException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/component/ComponentException.html), [CorrelationTimeoutException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/routing/correlation/CorrelationTimeoutException.html), [FailedToQueueEventException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/service/FailedToQueueEventException.html), [FilterUnacceptedException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/routing/filter/FilterUnacceptedException.html), [MessageRedeliveredException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/exception/MessageRedeliveredException.html), [ResourceNotFoundException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/transport/http/components/ResourceNotFoundException.html), [RestServiceException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/transport/http/components/RestServiceException.html), [RoutingException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/routing/RoutingException.html), [SecurityException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/security/SecurityException.html), [TransformerMessagingException](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/transformer/TransformerMessagingException.html)

|  |  |
| --- | --- |
| boolean | [**causedBy**](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/MessagingException.html#causedBy(java.lang.Class))([Class](http://java.sun.com/j2se/1.6.0/docs/api/java/lang/Class.html?is-external=true) e)            Evaluates if the exception was caused (instance of) by the provided exception type |
| boolean | [**causedExactlyBy**](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/MessagingException.html#causedExactlyBy(java.lang.Class))([Class](http://java.sun.com/j2se/1.6.0/docs/api/java/lang/Class.html?is-external=true) e)            Evaluates if the exception was caused by the type and only the type provided exception type i,e: if cause exception is NullPointerException will only return true if provided exception type is NullPointerException |
| boolean | [**causedRollback**](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/MessagingException.html#causedRollback())()            Signals if the exception cause rollback of any current transaction if any or if the message source should rollback incoming message |
| boolean | [**causeMatches**](https://www.mulesoft.org/docs/site/current/apidocs/org/mule/api/MessagingException.html#causeMatches(java.lang.String))([String](http://java.sun.com/j2se/1.6.0/docs/api/java/lang/String.html?is-external=true) regex)            Checks the cause exception type name matches the provided regex. |

Mule supports five types of messaging exception strategies, each of which is capable of handling errors that occur in flows which process transactions:

| **Exception Strategy** | **Use** | **Transaction Error Handling** |
| --- | --- | --- |
| [Default Exception Strategy](https://docs.mulesoft.com/mule-runtime/3.8/error-handling#default-exception-strategy) | Defined and implicitly applied by default to handle all messaging exceptions that are thrown in Mule applications | When a message throws an exception, the default exception strategy rolls back the message(if it transactional) and logs the exception. |
| [Catch exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/catch-exception-strategy) | Define a catch exception strategy to customize the way Mule handles any exception. Catch exception strategies consume inbound messages. | When a message throws an exception, the catch exception strategy always commits the transaction and consumes the message. |
| [Rollback exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/rollback-exception-strategy)(Support TX) | Define a rollback exception strategy to ensure that a message that throws an exception in a flow is rolled back for reprocessing (if the message source supports redelivery). | When a message throws an exception, the rollback exception strategy **makes one or more attempts to rollback the message and redeliver it for processing** (if the message source supports redelivery). If the message exceeds its redelivery attempts, then the rollback exception strategy takes the message from its inbound source and consumes the message. |
| [Reference exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/reference-exception-strategy) | Define a reference exception strategy to refer and adhere to the error handling parameters defined in a global catch, rollback or choice exception strategy. | When a message throws an exception, the reference exception strategy refers and adheres to the error handling parameters defined in a global catch, rollback or choice exception strategy. (The reference exception strategy itself never actually performs any rollback, commit, or consume activities.) |
| [Choice exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/choice-exception-strategy) | Define a choice exception strategy to customize the way Mule handles a message that throws an exception based on the message’s content at the moment it throws the exception. | When a message throws an exception, the choice exception **strategy makes a decision about where to route the message for** further processing. (The choice exception strategy itself never actually performs any rollback, commit, or consume activities.) |

**Characteristics of Messaging Exception Strategies**

* Each flow can contain only one exception strategy. However this can be a choice exception strategy that then refers to other nested exception strategies.
* Choice exception strategies can contain one or more catch and/or rollback exception strategies. (Rollback and catch exception strategies cannot, however, contain other exception strategies.)
* Each exception strategy can contain any number of message processors.
* The exception strategy message processors should not throw exceptions, because you cannot create a nested exception strategy for these message processors.

## Default Exception Strategy

Mule’s default exception strategy implicitly and globally handles all messaging exceptions that are thrown in Mule applications. You can override Mule’s default exception strategy by adding a catch, rollback, or choice exception strategy to a flow. Otherwise, when an exception is thrown in a flow, Mule automatically rolls back any pending transaction and logs the exception; if no transaction is involved, the default exception strategy simply logs the exception.

## Other Ways of Handling Errors

### Until Successful Scope

**Until Successful** behaves similarly to a rollback exception strategy. This scope attempts to route a message through its child flow until the message is processed successfully. However, you can define the maximum number of processing attempts the Until Successful scope undertakes before it reverts to handling the message as though it were an exception. You can configure a **Failure Expression**, an **Ack Expression**, or a **Dead Letter Queue Reference** to instruct the scope on how to manage messages that it cannot process. In this respect, Until Successful scope behavior is similar to a [rollback exception strategy](https://docs.mulesoft.com/mule-runtime/3.8/rollback-exception-strategy). Refer to the [Scopes](https://docs.mulesoft.com/mule-runtime/3.8/scopes) documentation for details.

### Exception Filter

Mule’s Exception filter stops normal flow execution when it discovers a message that contains a message in the exceptionPayload field. By comparison, an exception strategy typically stops normal flow execution when a message throws an exception in the flow. You can combine the two and configure the exception filter to stop normal flow execution and throw an exception, which triggers the exception strategy. Refer to the [Filters documentation](https://docs.mulesoft.com/mule-runtime/3.8/filters) for configuration details.

### Reconnection Strategies

Mule’s **Reconnection Strategies** specify how a connector behaves when its connection fails. You can control how Mule attempts to reconnect by specifying a number of criteria: the type of exception, the number and frequency of reconnection attempts, the notifications generated, and more. With a reconnection strategy, you can better control the behavior of a failed connection by configuring it, for example, to reattempt the connection only once every 15 minutes, and to stop trying to reconnect after 10 attempts. Reconnection strategy behavior resembles that of exception strategies, but reconnection strategies provide instructions specifically for — and limited to — reconnection attempts. Refer to the [Reconnection Strategies documentation](https://docs.mulesoft.com/mule-runtime/3.8/configuring-reconnection-strategies) for details.

### CXF Error Handling

Web services that utilize CXF can implement Mule exception strategies (such as the Catch and Rollback exception strategies) that are compatible with CXF. Consult the [CXF Error Handling documentation](https://docs.mulesoft.com/mule-runtime/3.8/cxf-error-handling) for details.

# Transaction Management

Transaction

**Atomicity**

All changes to data are performed as if they are a single operation. That is, either all happen or none happen

For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

**Consistency**

**Data is in a consistent state when a transaction starts and when it ends**.

For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

**Isolation**

**The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized**.

For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

**Durability**

**After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.**

For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

Mule TX:

When a series of steps in a flow must succeed or fail as one unit, Mule uses a transaction to demarcate such a unit. If all the steps execute successfully, TX will commit. Otherwise, rolled back to the earlier state. No any steps have partial updates of state.

## Configuring Transactions

You can demarcate a transaction by either applying a transactional configuration to a connector, or by wrapping several elements in a transactional scope.

* **Apply a transaction to an inbound connector** when you want Mule to handle the complete flow as a transaction.
* **Apply a transaction to an outbound connector** when you want Mule to handle the outgoing operation as part of an existing transaction.
* **Apply a transaction as a wrapper** (known as a **scope** in Studio) when you want to apply a transaction to elements within a flow that do not begin with a inbound connector configured as a transaction.

The following **connectors support transactional demarcation**:

* **JMS**
* **JDBC(Deprecate)**
* **VM**

## Transaction Types

Mule supports three different types of transactions:

<>**single resource**,

<>**multiple resource**,

<>**extended architecture (XA).**

Configure transactional attributes:

| **Attribute** | **Value** | **Available on Connector** | **Use** |
| --- | --- | --- | --- |
| **action** | NONE | JMS VM JDBC | When it receives a message, Mule resolves the transaction, then executes the operation as non-transactional. |
| ALWAYS\_BEGIN | JMS VM JDBC | When it receives a message, Mule always starts a new transaction. If a transaction already exists, Mule resolves the transaction. |
| BEGIN\_OR\_JOIN | JMS VM JDBC | When it receives a message, Mule joins a transaction if one is already in progress. Otherwise, Mule simply begins a new transaction. |
| ALWAYS\_JOIN | JMS VM JDBC | When it receives a message, Mule always expects a transaction to be in progress, and always joins the transaction. If no transaction is in progress, Mule throws an exception. |
| JOIN\_IF\_POSSIBLE | JMS VM JDBC | When it receives a message, Mule joins the current transaction if one is available. Otherwise, Mule does not begin a transaction. |
| NOT\_SUPPORTED | JMS VM JDBC | When it receives a message, this outbound connector executes *outside* the transactional operation; the transaction continues and does not fail. |
| **timeout** | - | JMS VM JDBC | Insert an integer to represent the number of milliseconds (ms) that Mule allows to pass before it ends the transaction. **Important:** The timeout transaction is only taken into account in XA transactions. |
| **interactWithExternal** | true | JMS VM JDBC | When set to true, Mule acknowledges transactions that began externally. For example, if you set the transaction action to BEGIN\_OR\_JOIN, and set interactWithExternal to true, Mule joins any transaction that is already in progress when it receives a message, regardless of whether the transaction began outside of Mule. |

**Configuration Tips and Tricks**

* Operations that occur inside a transaction execute **synchronously**. You cannot build an asynchronous flow inside a transaction.

# Rollback Exception Strategy

You can define a rollback exception strategy to ensure that a message that throws an exception in a flow is rolled back for reprocessing.

If the transaction fails, that is, if a message throws an exception while being processed, then the rollback exception strategy rolls back the transaction in the flow. If the inbound connector is transactional, Mule delivers the message to the inbound connector of the parent flow again to reattempt processing (that is, message redelivery).

Beyond managing transactional errors, you can use a rollback exception strategy to:

* Manage unhandled exceptions—​exceptions that the application fails to catch.
* Put in flows in which messages require redelivery.

A rollback exception strategy has the potential to introduce an infinite loop of activity within a flow: a message throws an error, the rollback exception strategy catches the exception and rolls the message back for reprocessing; the message throws an error again, the rollback exception strategy catches the exception again, and rolls the message back for reprocessing, and so on.

To avoid this infinite loop and responsibly manage unresolvable errors, you can apply two limitations to a rollback exception strategy:

* Define the maximum number of times that the rollback exception strategy attempts to redeliver the message for processing.
* Define a flow to handle messages that exceed the maximum number of redelivery attempts.

**Mule attempts message redelivery**

when your flow uses one of the following two types of transports:

<> **transactional** - **transport consumes a message as it travels through the flow**, transforming it to a different object, for example, or enriching it with more data. When a message using a reliable transport throws an exception, the rollback exception strategy discards the partially processed message and instructs the flow to attempt processing the original message again.

 <>**reliable(Processing Strategy -Synchronous)**.- **transport does not consume a message as it travels through the flow until it can ascertain that the message has successfully reached the end of the flow.** When a message using a reliable transport throws an exception, the rollback exception strategy discards the partially processed message and instructs the flow to attempt processing the original message again.

## Triggering Flows

The following table details the component to use in a flow to call other flows.

| **Type of Flow** | **Component** | **Execution Relative to Triggering Flow** | **Exception and Processing Strategies** |
| --- | --- | --- | --- |
| Subflow | Flow Reference | synchronous | inherited |
| Synchronous Flow(Private flow,execute on Same thread) | Flow Reference | synchronous | not inherited |
| Asynchronous Flow(execute on differentThread) | Flow Reference wrapped within an [Async Scope](https://docs.mulesoft.com/mule-runtime/3.9/async-scope-reference) | asynchronous | not inherited |

**Flow Processing Strategies:**

**Factors**

* 1. Flow’s exchange pattern
  2. Transactional

| **Exchange Pattern** | **Transactional?** | **Flow Processing Strategy** |
| --- | --- | --- |
| Request-Response | Yes | Synchronous |
| Request-Response | No | Synchronous |
| One-way | Yes | Synchronous |
| One-way | No | Queued-Asynchronous |

**Q.What are various types of Exception Handling?**

* 1. **Choice Exception Strategy.**

You can define a choice exception strategy to customize the way Mule handles a message with an error based on the message’s content at the moment it throws an exception. A choice exception strategy catches all exceptions thrown within its parent flow, examines message contents and exception type, then routes messages to the appropriate exception strategy for processing.

## When to Use

Use a choice exception strategy to enable Mule to make decisions about how to handle each error that occurs in a flow.

For example, in a flow that processes orders, you can use a choice exception strategy to apply the following error handling rules:

* Messages that throw an AlreadyProcessedException should be discarded.
* Messages that throw a ValidationException should be sent to an invalid order queue.
* All other messages which throw exceptions should be rolled back to retry processing.

Global choice Exception Strategy -

* Catch Exception Handling.
* Rollback Exception Handling.
* Global Exception Handling.
* Default Exception Handling.

**Q. What are the different types of Flow Processing Strategies?**  
There are six different types of Flow Processing Strategies. They are

* Asynchronous Flow Processing Strategy.
* Custom Processing Strategy.
* Thread Per Processing Strategy.
* Queued Asynchronous Flow Processing Strategy.
* Synchronous Flow Processing Strategy.
* Non-blocking Flow Processing Strategy.
* Queued Flow Processing Strategy.